

AD-A033 634

NAVAL UNDERSEA CENTER SAN DIEGO CALIF

F/G 9/2

PASCAL 1100.(U)

SEP 76 M S BALL

UNCLASSIFIED

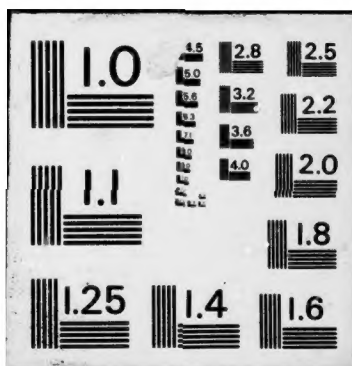
NUC-TP-527

NL

1 OF 1  
AD-A  
033 634



END  
DATE  
FILMED  
2-4-77  
NTIS



**U.S. DEPARTMENT OF COMMERCE  
National Technical Information Service**

AD-A033 634


PASCAL 1100

NAVAL UNDERSEA CENTER,  
SAN DIEGO, CALIFORNIA

SEPTEMBER 1976

ADA033634

365008

  
NUC TP 527



# PASCAL 1100

by

M. S. Ball

Fleet Engineering Department

September 1976



Approved for public release; distribution unlimited.

REPRODUCED BY  
NATIONAL TECHNICAL  
INFORMATION SERVICE  
U. S. DEPARTMENT OF COMMERCE  
SPRINGFIELD, VA. 22161





NAVAL UNDERSEA CENTER, SAN DIEGO, CA. 92132

---

AN ACTIVITY OF THE NAVAL MATERIAL COMMAND

**R. B. GILCHRIST, CAPT, USN**

Commander

**HOWARD L. BLOOD, PhD**

Technical Director

**ADMINISTRATIVE STATEMENT**

The work reported herein was sponsored by NAVELEX 316 and NAVSEA 06H1.

Released by

**R. H. HEARN, Head**  
Electronics Division

Under authority of

**D. A. KUNZ, Head**  
Fleet Engineering Department

**UNCLASSIFIED**

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER NUC TP 527	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle)  PASCAL 1100		5. TYPE OF REPORT & PERIOD COVERED Research and Development September 1975 - June 1976
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s)  M. S. Ball		8. CONTRACT OR GRANT NUMBER(s)  Task XF21.221.701.U011
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Undersea Center San Diego, CA 92132		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Electronics Systems Command National Center No. 2 Washington, D. C. 20360		12. REPORT DATE September 1976
		13. NUMBER OF PAGES 25
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)		15. SECURITY CLASS. (of this report)  Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT (of this Report)  Approved for public release; distribution unlimited.		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)		
18. SUPPLEMENTARY NOTES		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number)  PASCAL Programming languages UNIVAC 1110		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number)  This document describes the PASCAL 1100 compiler and runtime system. It defines those items left undefined in the language report by Jensen and Wirth and describes local extensions to the language. In addition, it describes procedures for the use of the language under EXEC VIII.		

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

**UNCLASSIFIED**

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

## CONTENTS

INTRODUCTION	3
1. HARDWARE REPRESENTATION	5
2. EXTENSIONS TO THE LANGUAGE PASCAL	5
2.1 Processor Format Call	6
2.2 External Procedures	6
2.2.1 Declaration	6
2.3 FORTRAN Procedures and Functions	7
2.4 Enhancements of "Read" and "Write"	8
2.5 Octal Numbers	9
3. SPECIFICATIONS LEFT UNDEFINED IN THE REPORT	9
3.1 The Program Heading and External Files	9
3.2 Standard Types	10
3.2.1 Integer	10
3.2.2 Real	10
3.2.3 Char	10
3.3 The Standard Procedure "write"	10
4. RESTRICTIONS	11
5. ADDITIONAL PREDEFINED IDENTIFIERS	12
5.1 Additional Predefined Constants	12
5.2 Additional Predefined Variables	12
5.3 Additional Predefined Procedures and Functions	12
5.3.1 Procedures	12
5.3.2 Functions	13
6. BRINCH HANSEN PASCAL	14
6.1 Extensions for Compatibility	14
6.2 The "B" Option	15
6.3 Incompatibilities	15
7. HOW TO USE THE PASCAL 1100 SYSTEM	16
7.1 Compiler Options	16
7.1.1 Compiler Directives	16
7.1.2 Control Card Options	17

## CONTENTS (Continued)

8. IMPLEMENTATION DETAILS	17
8.1 The Form of the Generated Code	17
8.2 Run-Time Organization	18
8.2.1 Memory Management	18
8.2.2 Procedure Entry and Exit	19
8.2.3 Register Usage	20
8.3 Diagnostic System	21
8.4 Representation of Files	21
8.5 Miscellaneous Topics	22
8.5.1 Standard Function References	22

## INTRODUCTION

This report describes the features of the Pascal implementation for the Univac 1100 series. The Pascal Language as a whole is described in *PASCAL - User Manual and Report*, by Kathleen Jensen and Niklaus Wirth.\* This report contains only those features which are peculiar to this implementation.

The Pascal compiler is also capable of compiling programs written in a Pascal dialect used by Per Brinch Hansen in his minicomputer-based implementation of Pascal. Those features related to this implementation are always available to the user, but are not generally available on other Pascal implementations. They will be described in a separate chapter.

This report is divided into eight sections:

1. Hardware representation
2. Extensions to the language.
3. Specifications left unspecified in the language definition.
4. Restrictions
5. Additional predefined procedures and functions.
6. Brinch Hansen Pascal
7. How to use Pascal 1100
8. Implementation details

\*Springer-Verlag, New York, 1974. *Lecture Notes in Computer Science* 18.

ACCESSION for	
NTIS	Write Section <input checked="" type="checkbox"/>
DOC	Buff Section <input type="checkbox"/>
UNANNOUNCED	<input type="checkbox"/>
JUSTIFICATION.....	
BY .....	
DISTRIBUTION/AVAILABILITY CODES	
Dist.	AVAIL. and/or SPECIAL
A	

## 1. HARDWARE REPRESENTATION

Pascal 1100 uses the ASCII character set exclusively. Within the compiler, lower case characters are converted into upper case before use as reserved words and identifiers, so that there is no difference between "begin" and "BEGIN" as far as the compiler is concerned.

All reserved words are written out without escape characters or underlining, for example

begin, end, case . . . .

Only the first 12 characters of any identifier are significant, and identifiers which differ only after the first 12 characters are considered identical.

The special symbols given in the manual are used as given, with the following additions.

Table 1-1. Alternate symbol representations.

standard	alternate
and	&
}	(*
}	*)

The arrow, used for pointers, is represented by "^".

All of the above alternates are in addition to the standard, and are included for compatibility with other implementations and the keypunch character sets.

As a final note, the character "-" (underline) is considered to be an alphabetic character within an identifier.

## 2. EXTENSIONS TO THE LANGUAGE PASCAL

This section describes non-standard language constructs available on the Pascal 1100 system. They are oriented toward the Exec VIII operating system and exist primarily to allow the utilization of the operating system facilities.

## 2.1 PROCESSOR FORMAT CALL

There are two basic formats for executing a program under Exec VIII. The most commonly used method for user programs is:

```
@XQT,OPTS  PROGRAM
```

This is the format with which a normal Pascal program should be executed.

The second format is used for processors and control cards, and is

```
@PROGRAM,OPTS  ARG1,ARG2,...
```

This method has the advantage of allowing the user to specify input and output files to the program on the calling card. It is also the format expected by the processor interface routines. A Pascal program may be constructed for this method of calling by replacing the program heading by the following.

```
<program> ::= <program heading> <block> . | <processor heading> <block> .  
<processor heading> ::= PROCESSOR <identifier> ( <process parameters> ) ;  
<process parameters> ::= <file specifier> { , <file specifier> }  
<file specifier> ::= <identifier> | <identifier> *
```

The <processor heading> generates a program which can be called using the second format above. The <file specifier> operates as usual, except that an asterisk (\*) following the file "INPUT" instructs the program to take its input from the system Source Input Routine (SIR). If input is from SIR, then the user also has access to the system routines for source output and relocatable output. This will be discussed later.

The INFOR table from the calling line may be manipulated using external procedures which are available in the library. See the Univac PRM for further data.

For example:

```
PROCESSOR prtd(input*,output);
```

specifies that the program will be called by a card

```
@PRTD  FILE.ELT
```

and will receive its input from the standard source input routine.

## 2.2 EXTERNAL PROCEDURES

The Pascal 1100 system allows the user to define and use externally compiled procedures.

### 2.2.1 Declaration

The declaration for an external procedure consists of a procedure heading followed by the word "EXTERN." For example:

```
PROCEDURE printit(tab: integer; ch: char); EXTERN;
```

```
FUNCTION gcd(x, y: integer): integer; EXTERN;
```

The user can define procedures to be called in this manner by inserting the reserved word "ENTRY" after the "PROCEDURE" or "FUNCTION" in the procedure heading. For example:

```
PROCEDURE ENTRY printit(tab: integer; ch: char); <body>
FUNCTION ENTRY gcd(x, y: integer): integer; <body>
```

Any procedure in the outer scope of the program may be declared an entry. In addition, a procedure or group of procedures may be compiled by themselves without a containing program. In this case, any variables declared outside of the procedures are available for use in the procedures and will be retained from call to call. File variables may not be declared in this outer scope, since they will not be initialized properly. A group of such procedures must be terminated with a period (".") just as a program is terminated. For example:

```
VAR i: integer;
PROCEDURE ENTRY initialize;
  BEGIN
    i := 0;
  END;
PROCEDURE ENTRY increment;
  BEGIN
    i := i + 1;
  END;
FUNCTION ENTRY current_value: integer;
  BEGIN
    current_value := i;
  END;
. {This period is necessary to terminate compilation}
```

### 2.3 FORTRAN PROCEDURES AND FUNCTIONS

FORTRAN procedures or functions can be declared in a manner similar to external procedures. To use a FORTRAN procedure, the user declares it in the same manner as a Pascal procedure, but replaces the body of the procedure with the word "FORTRAN." For example:

```
PROCEDURE plot(x, y: real; indx: integer); FORTRAN;
```

The compiler will generate the FORTRAN calling sequence for this procedure. Only code generated by the ASCII FORTRAN (FTN) processor may be used.

Although FORTRAN makes no distinction between value and variable parameters and will happily assign a value to an expression or constant, Pascal does make this distinction, and will make a local copy of any parameter declared to be value. This is done for safety reasons, and the programmer is advised to make use of this property whenever possible.

There are some incompatibilities in the representation of variables between Pascal and FORTRAN. The major one is the storage of arrays. Pascal considers multidimensional arrays to be arrays of arrays, which automatically defines storage with the rightmost of any group of subscripts varying more rapidly. This is called "rowwise storage." FORTRAN, on the other hand, uses columnwise storage, with the leftmost subscript varying most rapidly. This means that the user must transpose array elements in the Pascal code, or the FORTRAN program must take this into account.

The following table summarizes the type correspondence.

Table 2-1. FORTRAN vs Pascal types.

Parameter Type in FORTRAN	Parameter Type in Pascal	Remarks
INTEGER	integer	
REAL	real	
DOUBLE	none	double precision not supported in Pascal
COMPLEX	record re: real; im: real end;	
LOGICAL	Boolean	
ARRAY	ARRAY	see the note above for incompatibility
SUBROUTINE	PROCEDURE	formal procedures must be FORTRAN proc.
FUNCTION	FUNCTION	as procedure, also, result cannot be COMPLEX or DOUBLE, as record-valued functions are not allowed.

Only the above parameter types make sense if the external routine is actually written in FORTRAN. Note that the Pascal compiler does not check for legal argument types.

## 2.4 ENHANCEMENTS OF "READ" AND "WRITE"

The standard procedures "read" and "write" have been enhanced to work with all files, not just text files. The following equivalences now hold independent of file type.

```
write(f,v); <==> f^ := v; put(f);
read(f,v); <==> v := f^; get(f);
```

A further enhancement is the addition of octal editing when writing numbers to a character file. Octal editing is specified by writing the word "OCT" after the editing statement, thus:

```
write(i:4 OCT);
```

Octal editing is legal with all scalar or pointer variables and will write exactly the number of digits specified. If the value will not fit in the number of digits specified, it will be truncated from the left to fit. Any leading zeroes will also be printed. For example:

```
write(1024B:3 OCT); writes "024"
```

## 2.5 OCTAL NUMBERS

The user may specify octal numbers to the compiler by following an integer with the character "B". Thus:

1000B is equivalent to 512

## 3. SPECIFICATIONS LEFT UNDEFINED IN THE REPORT

### 3.1 THE PROGRAM HEADING AND EXTERNAL FILES

A Pascal file variable is implemented as a file under EXEC VIII. External files are those which are specified in the program heading, and must be assigned to the run at the time of program execution. Internal files are generated on entry to the procedure in which they are declared, and are released upon exit from that procedure.

There are some predeclared files which may be used in the program heading. These allow the user access to EXEC VIII standard files.

**Input:** text; This is the standard input file maintained by all programs. It is the file obtained by the normal system READ routine. To allow the use of Pascal programs with Demand terminals, this file is initialized with EOLN true. To obtain the first image, the user must issue a "read". A "reset" on this file has no effect.

**Input\*** :text; When followed by an asterisk, this file obtains its data from the source input routine. In this case, a "reset" will close out the current pass and begin the next pass over the input data. Since the user cannot use the source or relocatable output routines until after the first pass is completed, this is quite important. In all other ways, the performance is identical to the normal use of "Input." See the Univac PRM for more data.

**Output:** text; The output file writes to the standard output stream using the system "PRINT" command. It must be declared in any program. A call to "rewrite" has no effect on this file.

**SOR:** text; This writes a symbolic element using the system symbolic output routine (SOR). This is meaningful only when a processor call card is used. The file must be initialized with a "rewrite" after the first pass of SIR

(if used) is complete. Subsequent "rewrite"s to this file will produce an error termination. See the Univac PRM for more data.

**Punch:** text; This produces punched cards using the system PUNCH command. A call to "rewrite" will have no effect. Note that the maximum record length for this file is 80 characters. The ASCII characters written to this file will be converted to their fielddata equivalents before punching.

### 3.2 STANDARD TYPES

#### 3.2.1 Integer

The standard type "integer" is implemented with the standard 1100 series integer word, which allows values in the range -34359738367..34359738367.

The standard constant "MAXINT" is set to 34359738367.

Warning: check for integer overflow except on divide is impractical on the Univac 1100 series. It is the users responsibility to make sure that these limits are not exceeded.

#### 3.2.2 Real

Real numbers are implemented using the 1100 single-precision floating-point format. This allows 27 bits for fraction, which corresponds to about 8 significant decimal digits. The limits on the exponent are between -39 and 38. Exceeding the upper limit causes a run-time error, and dropping below the lower limit causes the result to be set to a true zero.

#### 3.2.3 Char

The type "char" contains all of the standard ASCII characters, including unprinting control characters. Although ASCII is a 7 bit code, with possible ordinate values between 0 and 127, the 1100 series typically allows 9 bits for each character and stores them 4 to a word. The Pascal system also follows this practice, and a packed array of char will have 4 characters per word, each with 9 bits of space allocated.

### 3.3 THE STANDARD PROCEDURE "write"

If no minimum field length parameter is specified, the following default values are assumed.

Table 3-1.

Type	Default
integer	12
real	14 (where the exponent is always written E+xx)
Boolean	12
char	1
a string	Length of the string
octal format	12

If the length of a line will exceed the allowed length for the file (80 for punch, 132 otherwise), the line will be terminated and a new line written. The effect is the same as if "writeln" had been called. There are no spacing control characters at the start of each line, and all of the characters in a line are printed.

The standard data format for text files truncates trailing blanks, then pads the number of characters to a multiple of 4 with blanks. This means that if a file is written, then read, the number of blanks read at the end of line may not be the same as the number written.

#### 4. RESTRICTIONS

- 1) The words "entry", "processor", and "univ" (used by Brinch Hansen style programs) are reserved.
- 2) The base type of a set must be
  - a) a scalar with at most 143 elements (including char)
  - b) a subrange with a minimum element greater than or equal to 0 and a maximum element less than or equal to 143.
- 3) Standard functions or procedures cannot be used as actual procedure parameters. For instance, to run program 11.6 from the manual, one would have to write auxiliary functions as follows

```

. . . .
function sine(x: real): real;
  begin sine := sin(x) end;
function cosine(x: real): real;
  begin cosine := cos(x) end;
function zero(function f: real; a,b: real): real;
  begin ... end;
. . .
begin
  read(x,y); writeln(x,y,zero(sine,x,y));
  read(x,y); writeln(x,y,zero(cosine,x,y));
end.
```

- 4) It is not possible to construct a file of files; however, records and arrays with files as components are allowed.
- 5) It is not possible to declare files in a dummy outer block containing external procedures, as the files will not be initialized properly.

## 5. ADDITIONAL PREDEFINED IDENTIFIERS

### 5.1 ADDITIONAL PREDEFINED CONSTANTS

The constant "linenumber" is predefined. This is an integer whose value is always the current linenumber in the source program. Although the value changes, the identifier is treated as a constant by the compiler. This is useful in debugging for inserting error messages which refer to the source line.

### 5.2 ADDITIONAL PREDEFINED VARIABLES

The variable "options" is defined by:

options: set of char;

This variable is allocated in every program, and is initialized by the system using the options specified on the @ XQT or the processor call line. The character corresponding to each option character will be included in the set. Note that the system considers all options to be upper case, no matter what is typed.

### 5.3 ADDITIONAL PREDEFINED PROCEDURES AND FUNCTIONS

#### 5.3.1 Procedures

halt(message); Writes the string "message" to the print file and initiates an error walkback. This is primarily for use in library procedures written in Pascal.

mark(p); Sets the pointer "p" (which may be any pointer type) to the current heap top. This may be used with the procedure "release" below.

release(p); Resets the current heap top to the value in the pointer "p". (which may be of any pointer type) This value should have been set by the procedure "mark" above. These are primitive routines which allow the user to simulate a second stack on the heap. Since the routine "dispose" is presently not functional, this is the only method available to return storage to the heap.

The following procedures are for programmers interfacing closely with the executive, and are of little interest to the average programmer. In all cases, further details are available in the Univac PRM.

srer(kbits); (applicable only for processor) This opens the system relocatable output routine (ROR) for the generation of relocatable code. The integer "kbits" sets the "kbit limit" for the element.

ror(pkt); (applicable only for processor) This writes a word to the relocatable output. "pkt" is any array which contains the relocation data and word to write.

eror(trans,translc); (applicable only for processor) This closes ROR and sets up the transfer address. "Trans" and "translc" are the relative address and location counter of the transfer location. If no transfer location is desired, "trans" should be negative.

tblwr(table,length); (applicable only for processor) Writes a relocatable preamble in the array "table" of length "length" to the relocatable element. ROR must be closed. For more data, see the Univac PRM.

er(index, "A0","A1"); (Warning — use of this procedure may be hazardous to your sanity). This procedure is very low level, and provides no protection for error. It does allow the user access to the EXEC VIII "ER" mechanism from Pascal, an ability which is sometimes necessary. The "index" is a constant which is the ER index. It is checked against an internal set of allowed values before the call is generated. "A0" and "A1" are optional integer variables which, if they exist, are loaded into the hardware registers A0 and A1 before the call. On return from the ER, the final values of A0 and A1 are stored in these variables. The function "address", described below, allows the use of this procedure with Pascal variables for the packets.

Allowed ER index values are given in the following table.

Table 5-1. Allowed ER index values.

Name	Val	Name	Val
ABORT\$	012B	ACLIST\$	141B
ACSF\$	140B	APCHCN\$	075B
APRTC\$	074B	COND\$	066B
DATE\$	022B	EABT\$	026B
ERR\$	040B	EXIT\$	011B
EXLINK\$	173B	FACIL\$	114B
FACIT\$	143B	FITEM\$	032B
INFO\$	116B	IO\$	001B
IOARB\$	021B	IOW\$	003B
LABEL\$	031B	LINK\$	171B
PCT\$	064B	PFD\$	106B
PF1\$	104B	PFSS\$	105B
PFUWL\$	107B	PFWL\$	110B
RLINK\$	172B	RLIST\$	175B
SETC\$	065B	TDATE\$	054B
TIME\$	023B	TSWAP\$	135B
TWAIT\$	060B	WAIT\$ *	006B
WANY\$	007B		

\*Because of the unusual calling sequence of WAIT\$, it generates:

```

L   A0,"A0"
TP  3,A0
ER  WAIT$
S   A0,"A0"

```

### 5.3.2 Functions

conv(i); (Used by Brinch Hansen programs.) Returns a real value for the integer "i". For use in Brinch Hansen Pascal, which has no implicit conversion to real.

The following functions are primarily for those programmers writing code which interfaces closely with the executive or the hardware, and should be of little interest to the average programmer. More details are available in the Univac PRM.

expo(x); Returns an integer equal to the exponent part of the real number "x".

This is in excess 64 notation.

address(x); (Warning – use of this procedure may be hazardous to your sanity!)

Returns an integer value equal to the machine address of the variable "x", which may be of any type. This is provided to allow use of the standard procedure "er", and is of no use otherwise.

## 6. BRINCH HANSEN PASCAL

The compiler will accept the dialect of Pascal used by Per Brinch Hansen and called "Sequential Pascal" by him. For a complete description, see his publication.\* The Pascal 1100 Implementation makes no attempt to follow the same restrictions as that dialect, but the following extensions allow the compilation of programs written in it. There is no guarantee that programs which run under this system will also be legal under Brinch Hansen's system.

### 6.1 EXTENSIONS FOR COMPATIBILITY

The following extensions are available at all times.

- 1) The following symbols may be used as alternate representations of standard symbols:

Table 6-1. Alternate symbol representations.

Standard	Alternate
[	(.
]	.)
^	@
{	"
}	"

- 2) The reserved word "UNIV" is allowed in parameter lists. The modified syntax is.

⟨parameter group⟩ ::= ⟨identifier⟩ {, ⟨identifier⟩} : ⟨parameter type⟩

⟨parameter type⟩ ::= ⟨type identifier⟩ | UNIV ⟨type identifier⟩

The "UNIV" before the type identifier informs the compiler that any parameter type is acceptable as long as it has the same size as the format type. This is obviously highly machine dependent.

\*Per Brinch Hansen and Alfred C. Hartman, *Sequential Pascal Report*, Information Science, Calif. Inst. Tech. July, 1975.

- 3) The procedure "conv" converts integer to real explicitly.
- 4) The boolean operators "and" and "or" will accept set operands. When they are used, "and" is equivalent to "★" and "or" is equivalent to "+".
- 5) A constant string of any length may be used as a value parameter or in an assignment statement to any packed array of characters. The lower subscripts are automatically aligned, and the string is truncated or extended with nulls to match the length of the array.

- 6) Within a string or character constant the syntax

⟨ordinate expression⟩ ::= ( : ⟨number⟩ : )

may be used. This inserts into the string at that point a character whose ordinate is the value of ⟨number⟩.

- 7) Within an expression or a "WITH" statement, a pointer-valued function may be used in place of a pointer variable. For example:

type realp = ^real;  
FUNCTION p(x: real): realp;

.....

y := p(3.0)^ + 5.0;

This is forbidden by the Pascal Report and Brinch Hansen's own report, but is used extensively in his code.

## 6.2 THE "B" OPTION

The "B" option to the compiler (see the chapter on compiler use) specifies that the program being compiled is to be treated as a Brinch Hansen program. This makes the following changes in the form of the source program and the compiled code.

- 1) A "prefix" is now allowed. This contains type, constant, and procedure/function declarations. The type and constant declarations are entered into the outer scope of the program, and the procedures and functions are declared as external. This allows the direct simulation of Brinch Hansen's operating system interface through the prefix.
- 2) The main program will be compiled as an entry procedure which can be called from another program. In this case, the program argument list can have the same form as a procedure parameter list, rather than containing external file names.
- 3) All arrays of characters are treated as packed arrays.

## 6.3 INCOMPATIBILITIES

There are many incompatibilities between Pascal 1100 and the Brinch Hansen compiler. Rather than list them all, the user is referred to the publication above. The following are a few of the less obvious problems.

- 1) Brinch Hansen style compiler options are not accepted by Pascal 1100.
- 2) In Brinch Hansen Pascal, a case selector value with no corresponding statement is ignored. In Pascal 1100, it produces a run-time error.
- 3) Pascal 1100 does not initialize pointers to nil.
- 4) Brinch Hansen Pascal uses constant parameters while Pascal 1100 uses value parameters. The difference is that no value can be assigned to a constant parameter within a procedure, while a value parameter can be treated as a local variable.

## 7. HOW TO USE THE PASCAL 1100 SYSTEM

The Pascal system consists of a compiler, which converts the Pascal code into Univac 1100 series relocatable code, and a run-time library, which provides utility and I/O functions for the compiled code. The location of the compiler and run-time library is dependent upon the local system configuration. Check with the computer center for your site.

The Pascal compiler (PAS) is a Pascal program. It is a processor, and is called in the standard manner. The processor call statement is:

`@PAS,OPTS SOURCE,RELOC,UPDATEDSOURCE`

When the program is compiled, the relocated elements must be collected with the Pascal run-time library, and the program can then be executed in the usual manner. Any external files must be assigned at the time of execution.

### 7.1 COMPILER OPTIONS

The behavior of the compiler may be varied by the use of certain options. There are two types of options recognized by the Pascal compiler, control card options, specified on the calling control card, and compiler directives included in the code.

A compiler directive is written as a special form of comment with a \$-character as the first character

`{$(option sequence) (any comment)}`

for example:

`{SS+,T- this sets "S" and resets "T"}`

normally, a "+" following an option will activate that option and a "-" will deactivate it.

#### 7.1.1 Compiler Directives

The following options operate as compiler directives.

- A Generate code to check the values assigned to variables of subrange type to make sure that they are within bounds.

default = A+

- B** Generate a Brinch Hansen style program  
default = B-
- L** Generate a full compiler listing including generated code.  
default = L-
- R** If one or two digits follow the R, this is the amount of extra space allocated for dynamic variables (those not declared in the main program). The number specifies this in thousands of words.  
default = R2      2000 extra words
- S** Produce a source listing  
default = S-
- T** Include run-time tests for subscripts out of range.  
default = T+
- Z** Include line number diagnostics for error termination.  
default = Z+

Most of these compiler directives can be activated and deactivated as often as required, and so can be applied selectively to different portions of the code. "B" and "R" are obvious exceptions.

### 7.1.2 Control Card Options

Control card options follow:

- B** Set B+. See above.
- I** Inserting a new element from the run stream
- L** Set L+. See above.
- O** Set A-,T- (omit tests). See above.
- S** Set S+. See above.
- Z** Set Z-. See above.

## 8. IMPLEMENTATION DETAILS

### 8.1 THE FORM OF THE GENERATED CODE

The Pascal compiler generates code in the standard relocatable format, using the following location counters:

- 1 Procedure code
- 2 Literals

- 3 Forward reference links
- 4 Working storage

Working storage is allocated at compile time, and includes space for all variables allocated at the global level, plus the reserve specified with the "R" compiler directive. At the moment, this is the entire storage allocation for both the stack and the heap. There are plans underway to remove this restriction and allow the dynamic increase of this space up to 262K. The code generated by the compiler will support this, but the run-time system does not.

Forward references are handled by assigning a location under location counter 3 at the time of the reference. When the reference is resolved, the procedure or label location is entered into a table, and at the end of the program code is emitted under location counter 3 to complete the references.

## 8.2 RUN-TIME ORGANIZATION

### 8.2.1 Memory Management

The working store is used for both the stack and the heap.



Figure 8-1. Working storage allocation.

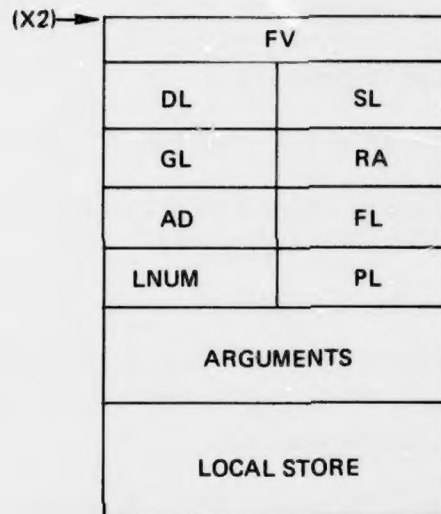
Note that X1 is used to point to the global area, even though this is not strictly necessary. This was done because it simplifies the implementation of Brinch Hansen style programs (callable as procedures) and very seldom adds anything to the cost of other programs. If this proves to be a significant overhead, it may be eliminated in future implementations.

Upon entry to an external procedure, X1 is set to point to its own global storage, but local storage for the procedure is put on the same stack as other procedures.

The storage handling on procedure entry and exit is simple and conventional, and best seen by examining the library procedures which handle it.

### 8.2.2 Procedure Entry and Exit

Each Pascal procedure called has space allocated on the top of the stack for its activation record. This has the following form:



- FV — FUNCTION RETURN VALUE
- DL — DYNAMIC LINK (X2 IN SURROUNDING ENV.)
- SL — STATIC LINK (X2 IN STATIC ENV.)
- GL — GLOBAL LINK (X1 IN SURROUNDING ENV.)
- RA — RETURN ADDRESS
- AD — ALLOCATION DATA (OLD STACK TOP)
- FL — FILE LINK (FOR CLOSING FILES)
- LNUM — LINE NUMBER OF CALL
- PL — PROCEDURE LOCATION FOR DIAG.

Figure 8-2. Activation record.

The procedure calling sequence generated by the compiler is as follows:

```

SX  "static link",SL
SX  X2,DL.
.
"compute parameters"
.
AX  X2,"local length",.U . NEW X2
LMJ X11,PROC
"return location"

```

Figure 8-3. Procedure call.

The code generated in the procedure has the form:

```
PROC      'proc                      . for diagnostics
          LMJ X10,P$ENTRY
          +  "argument length","total space"
          "code to open files"
          "code to store arguments"
          .
          "procedure code"
          .
          LMJ X10,P$EXIT
```

Figure 8-4. Procedure code form.

The details of filling in the linkage data and for initializing and closing files is best seen by examining the library routines for these functions.

Arguments may be considered to be in three categories

- direct     Those value arguments which will fit in a single word.
- indirect   Variable arguments, or value arguments too large to fit in a single word.
- setvalue   Set arguments passed by value.

In general, arguments are passed in registers whenever possible, with the called procedure storing them in its local activation record. When the number of arguments exceeds the available registers, they are stored into the activation record by the calling program. The compiler allocates space in the activation record as follows:

- direct     A single word in the argument area to hold the value.
- indirect   A single word in the argument area to hold the address. In addition, value arguments have enough space allocated in the local variable area to hold a local copy of the argument.
- setvalue   Sufficient space (4 words currently) to hold the set value.

The arguments are passed as follows:

- direct     Up to 9 arguments are passed in the registers A4 to A12, in that order. Others are passed in core.
- indirect   Up to 4 arguments are passed in the registers X8 down to X5, in that order. Others have their addresses stored in core.
- setvalue   Always passed in core.

### 8.2.3 Register Usage

The registers X1, X2, and R10 through R15 are used in storage allocation and must be left undisturbed. All other registers are available for use.

### 8.3 DIAGNOSTIC SYSTEM

The diagnostic system is about the minimum livable. On error, it provides an error message and a walkback through the stack, listing calling lines and procedure names. In most cases, this provides the data necessary for fault location.

The diagnostic system works by keeping the current line number in the register R10. This line number, plus the address of the called procedure are kept in the stack on procedure call. At the start of any procedure, the compiler inserts the procedure name in ASCII. This allows a simple trace of the calls, and a reasonably readable walkback. For code generated without diagnostic data, the walkback system gives diagnostics in terms of octal locations.

To avoid ambiguities with partially constructed activation records, the address of the latest currently completed mark stack is kept in the location P\$CMARK.

### 8.4 REPRESENTATION OF FILES

Text files are represented in the system standard "SDFF" file format. These can be read by the majority of the system processors, such as the editor. The Pascal system can read files produced by any system processor, by FORTRAN formatted write statements, or by a symbiont. If the file is in fielddata code, it will be translated to ASCII by the file handler. The maximum record length allowed is 132 characters.

Any other data file is written in a system chosen format with a block length which is an integral number of sectors long. The format of each record is:

ELTSIZE	NELT
DATA	

NELT: NUMBER OF ELTS IN THIS BLOCK

ELTSIZE: IF POSITIVE, ELEMENT SIZE, IF  
NEGATIVE, ELEMENTS PER  
WORD.

Figure 8-5. Binary file format.

The length of the block, except for the last block in the file, can be computed from  $nelt * eltsiz$  rounded up to the next multiple of 112 words.

The end of file is denoted by a block with  $ELTSIZE < 0$ . On tape, a hardware file mark will follow to allow copying using FURPUR.

## **8.5 MISCELLANEOUS TOPICS**

### **8.5.1 Standard Function References**

References to the standard functions such as "sin" and "cos" generate a "LIJ" reference to the Mathematical Function common bank "RMATH\$". This is optimum for the 1110, but involves considerable overhead for other members of the 1100 series (approximately 151 microseconds on the 1108). If the system is installed on a machine other than an 1110, and these functions are expected to see heavy use, it would pay to change the compiler to generate "LMJ" references.